Department of Computer Science & Engineering
IIT Guwahati, India, March 12, 2018

CS347 Compilers Lab - Assignment 2

Ayush soni (150101014)
Mayank agrawal (150101033)
Abhishek kumar (150101003)

1) LANGUAGE SUPPORT:

We have created a grammar for a basic language supporting following specifications:
• Global declarations for class,functions and variables.
• Arithmetic expressions (+, -, *, /, %)
• Paranthesis (,),{,}
• Unary operator &, |, ++, --
• Relational operators
• Variable types – bool, int, string and int arrays
• Loop construct – while,for
• Conditional statement - if else,if
• Nesting of loops and conditions are allowed
• Type checking, paranthesis matching , arithmetic operator's precedence ensured
• Recursion allowed
• Variable declaration , function declaration inside class and constructors allowed.
• function can return some values or it can be void.

2) GRAMMAR (PRODUCTION RULES)

Following are the production rules of context free grammar for the language mentioned above.

A context-free grammar G can be mathematically defined as:
G = (V,X, R, S) where,
V = set of non-terminals,
P = set of terminals,
R = production rules, and
S = start symbol.

- Program -> declaration_list | statement_list
- declaration_list -> declaration declaration_list | ep.
- declaration -> class_decl | var_decl | func_decl

Above productions define the global declaration for variables, classes and functions

- var_decl -> var_type ID DEL | var_type ID[NUM] DEL
- var_type -> int | string | float | bool

Above productions define the variable declaration as int, string, float, bool type

- class_decl -> classID{class_body} DEL
- class_body -> specifier_list func_list

- specifier_list -> var_decl specifier_list | var_decl
- func_list -> func_decl func_list | ep.

Above productions define class declaration with variable list and function list

- func_decl -> var_type ID(parameter_list){func_body} DEL
- parameter_list -> parameter | parameter more_parameter | ep.
- more_parameter -> , parameter | , parameter more_parameter
- parameter -> var_type ID | var_type ID[NUM]
- func_body -> statement_list return expression DEL | statement_list

Above productions define function declaration with parameters and return statement

- statement_list -> statement DEL statement_list | ep.
- statement -> var_decl | loop | ifstatement | func_call | class_object | expression

Above productions define type of possible statements

- loop -> for_loop | while_loop
- for_loop -> for (expression SEMI condition SEMI expression) {statement_list}
- while_loop -> while(condition){statement_list}
- ifstatement -> if condition {statement_list} else {statement_list} | if condition {statement_list}

Above productions define loop construct (while & for), conditional statement (if-else) and all types of nesting and recursion

- func_call -> var = call| call
- var -> ID | ID[NUM] | ID[ID]
- call -> object_func_call | normal_func_call
- object_func_call -> var.normal_func_call
- normal_func_call -> ID(passed_values)
- passed_values -> value more_values | ep.
- more_values -> , value | , value more_values
- value -> val | val_list
- val_list -> [val more_val]
- more_val -> , val | , val more_val
- val -> BOOL | “STRING” | ‘STRING’ | ID | NUM | FLOAT

Above productions define function call with all types of parameters

- class_object -> var = classID(asignment_list)
- assignment_list -> assignment more_assignment | ep.
- more_assignment -> , assignment | , assignment more_assignment
- assignment -> ID ASSIGN val | ID ASSIGN [val_list] | [val_list] | val | ep.

Above productions define constructor declaration for a class

- expression -> ID = condition | ID = arithmatic_op | condition
- condition -> arithmatic_op COMPARATOR arithmatic_op | UNARY_OP arithmatic_op | arithmatic_op UNARY_OP
- arithmatic_op -> mul SUM arithmatic_op | mul
- mul -> f MULT mul | f
- f -> ID | (arithmatic_op)

Above productions define all type of possible expressions supported by our language

- **NUM** -> [0-9]*
- **FLOAT ->** **NUM.NUM**
- **STRING** -> [a-zA-Z0-9_]*
- **BOOL** -> true | false
- **DEL** -> /n
- **ID** -> [a-z][A-Za-z0-9_]*
- **classID** -> [A-Z][A-Za-z0-9]*
- **SEMI** -> ;
- **ASSIGN** -> : | =
- **COMPARATOR** -> < | > | == | <= | >= | !=
- **SUM** -> + | -
- **MULT** -> * | /
- **UNARY_OP** -> ! | ++ | --

Above productions are the list of all terminals

3) LEXER:

All lexemes of the grammar consist of combinations of above mentioned terminals (see the production rules all the terminals are indicated in bold capital letters) . Hence, tokens of the grammar can be defined using these following sets.
The token classes are defined as follows:

1. KEYWORD = int | break | return | if | else | while | for | float | bool
2. BRACKETS = { ,(, [,},),,]
3. IDENTIFIER = string starting with a small letter
4. OPERATOR = sum_op | multiply_op | Assign_op | logical_op | relational_op | unary_op
5. WHITESPACE = \t
6. FLOATCONST = for floating point integers
7. NUMCONST= for integers
8. STRINGCONST= text enclosed with in “---” or ‘-----’
9. ARRAY_ID = for integer arrays
10. BOOLCONST = true | flase
11. DELIMITER = \n, ;, , ,.
12. CLASS_ID = string starting with a capital letter

4) Lexcode for the above mentioned tokenizer:

DIGIT [0-9]
TEXT_NUMBERS [a-zA-Z0-9_]

```
STRING                  [a-zA-Z0-9_ ]
NUM                     {DIGIT}+
DOT                     "."
FLOAT                   {NUM}{DOT}{NUM}
BOOLCONST               "true"|"false"
STRING1                 \"{STRING}*\"
STRING2                 \'{STRING}*\'
ASSIGN                  ":"|"="
ARITHMETIC_OP           SUM_OP|MUL_OP|ASSIGN
SUM_OP                  "+"|"-"
MUL_OP                  "*"|"/"
LOGICAL_OP              "&"|"|"
UNARY_OP                "!"|"++"|"--"
RELATIONAL_OP           ">"|"<"|">="|"<="|"=="|"!="
IDENTIFIER              {ID}|{ARRAY_ID}|{CLASS_ID}
CLASS_ID                [A-Z]{TEXT_NUMBERS}*
ARRAY_ID                [a-z]{TEXT_NUMBERS}*"["{NUM}"]"|[a-z
                        {TEXT_NUMBERS}*"["{ID}"]"
ID                      [a-z]{TEXT_NUMBERS}*
KEYWORD                 "int"|"bool"|"string"|"float"|"print"|"read"|"if"|"else"|"while"|"for"
DELIMITER               "\n"|";"|","|"."
WHITESPACE              [ \t]
BRACKETS                "{"|"}"|"["|"]"|"("|")"

%%

{FLOAT}                         { printf("<FLOATCONST,%s>\n", yytext); }
{NUM}                           { printf("<NUMCONST,%s>\n", yytext); }
{STRING2}                       { printf("<STRINGCONST,%s>\n", yytext); }
{STRING1}                       { printf("<STRINGCONST,%s>\n", yytext); }
{BOOLCONST}                     { printf("<BOOLCONST,%s>", yytext); }
{DELIMITER}                     { printf("<DELIMITER,%s>\n", yytext); }
{KEYWORD}                       { printf("<KEYWORD,%s>\n", yytext); }
{CLASS_ID}                      { printf("<CLASS_ID,%s>\n", yytext); }
{ARRAY_ID}                      { printf("<ARRAY_ID,%s>\n", yytext); }
{ID}                            { printf("<IDENTIFIER,%s>\n", yytext); }
{SUM_OP}                        { printf("<OPERATOR,%s>\n", yytext); }
{MUL_OP}                        { printf("<OPERATOR,%s>\n", yytext); }
{ASSIGN}                        { printf("<OPERATOR,%s>\n", yytext); }
{LOGICAL_OP}                    { printf("<OPERATOR,%s>\n", yytext); }
{UNARY_OP}                      { printf("<OPERATOR,%s>\n", yytext); }
{RELATIONAL_OP}                 { printf("<OPERATOR,%s>\n", yytext); }
{BRACKETS}                      { printf("<BRACKETS,%s>\n", yytext); }
{WHITESPACE}                    { printf("<WHITESPACE,%s>\n", yytext); }

%%

int main(int argc, char *argv[]) {
        yylex();
        return 0;
}
```

5) EXAMPLE:

INPUT:

```
job_1 = Job(job_id=1, flops_required = 100, deadline = 200, mem_required = 1024,affinity = [0.2,0.5,1,2])

ram = Memory(memory_type= 'primary', mem_size = 2048,, name = "ram1")

while(! Ram.get_available_memory())
{
wait(1)
}

if job_1.get_memory() <= ram.get_available_memory()
{
proc_1.submit_jobs(job_1)
}
else
{
discard_job(job_1)
}
```

OUTPUT:

```
<IDENTIFIER,job_1>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<CLASS_ID,Job>
<BRACKETS,(>
<IDENTIFIER,job_id>
<OPERATOR,=>
<NUMCONST,1>
<DELIMITER,,>
<WHITESPACE, >
<IDENTIFIER,flops_required>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<NUMCONST,100>
<DELIMITER,,>
<WHITESPACE, >
<IDENTIFIER,deadline>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<NUMCONST,200>
<DELIMITER,,>
<WHITESPACE, >
<IDENTIFIER,mem_required>
<WHITESPACE, >
```

<OPERATOR,=>
<WHITESPACE, >
<NUMCONST,1024>
<DELIMITER,,>
<IDENTIFIER,affinity>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<BRACKETS,[>
<FLOATCONST,0.2>
<DELIMITER,,>
<FLOATCONST,0.5>
<DELIMITER,,>
<NUMCONST,1>
<DELIMITER,,>
<NUMCONST,2>
<BRACKETS,]>
<BRACKETS,)>
<DELIMITER,
>
<DELIMITER,
>
<IDENTIFIER,ram>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<CLASS_ID,Memory>
<BRACKETS,(>
<IDENTIFIER,memory_type>
<OPERATOR,=>
<WHITESPACE, >
<STRINGCONST,'primary'>
<DELIMITER,,>
<WHITESPACE, >
<IDENTIFIER,mem_size>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<NUMCONST,2048>
<DELIMITER,,>
<DELIMITER,,>
<WHITESPACE, >
<IDENTIFIER,name>
<WHITESPACE, >
<OPERATOR,=>
<WHITESPACE, >
<STRINGCONST,"ram1">
<BRACKETS,)>
<DELIMITER,
>
<DELIMITER,
>

<KEYWORD,while>
<BRACKETS,(>
<OPERATOR,!>
<CLASS_ID,Ram>
<DELIMITER,.>
<IDENTIFIER,get_available_memory>
<BRACKETS,(>
<BRACKETS,)>
<BRACKETS,)>
<DELIMITER,
>
<BRACKETS,{>
<DELIMITER,
>
<IDENTIFIER,wait>
<BRACKETS,(>
<NUMCONST,1>
<BRACKETS,)>
<DELIMITER,
>
<BRACKETS,}>
<DELIMITER,
>
<DELIMITER,
>
<KEYWORD,if>
<WHITESPACE, >
<IDENTIFIER,job_1>
<DELIMITER,.>
<IDENTIFIER,get_memory>
<BRACKETS,(>
<BRACKETS,)>
<WHITESPACE, >
<OPERATOR,<=>
<WHITESPACE, >
<IDENTIFIER,ram>
<DELIMITER,.>
<IDENTIFIER,get_available_memory>
<BRACKETS,(>
<BRACKETS,)>
<DELIMITER,
>
<BRACKETS,{>
<DELIMITER,
>
<IDENTIFIER,proc_1>
<DELIMITER,.>
<IDENTIFIER,submit_jobs>
<BRACKETS,(>
<IDENTIFIER,job_1>
<BRACKETS,)>
<DELIMITER,

>
<BRACKETS,}>
<DELIMITER,
>
<KEYWORD,else>
<DELIMITER,
>
<BRACKETS,{>
<DELIMITER,
>
<IDENTIFIER,discard_job>
<BRACKETS,(>
<IDENTIFIER,job_1>
<BRACKETS,)>
<DELIMITER,
>
<BRACKETS,}>

<DELIMITER,
>
<IDENTIFIER,discard_job>
<BRACKETS,(>
<IDENTIFIER,job_1>
<BRACKETS,)>
<DELIMITER,
>
<BRACKETS,}>

6) PROCESS SCHEDULER:

```
#include<stdio.h>

int main(){
   int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
   printf("Enter total number of processes(maximum 20):");
   scanf("%d",&n);

   printf("\nEnter Process Burst Time\n");
   for(i=0;i<n;i++){
      printf("P[%d]:",i+1);
      scanf("%d",&bt[i]);
   }

   wt[0]=0;    //waiting time for first process is 0

   for(i=1;i<n;i++){
      wt[i]=0;
      for(j=0;j<i;j++)
         wt[i]+=bt[j];
   }

   printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
```

```
    for(i=0;i<n;i++){
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;
    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d\n",avtat);

    return 0;
}
```